Mark Palladino, Joshua Navarro (and originally Darrion Burgess, but he dropped the class)
CMPS 109 Fall 2016
MIS Project Report

Build instructions:

run “make” to build misvm
run “make clean” to remove misvm and all .out and .err files

Design Decisions:

What follows is all the parts of the program and our design decisions for how to implement each part.

Handling of errors

We have a struct called Error that has an error code prefix, an error code, and a message. All exceptions are thrown with this struct. In most cases, the error code is a line number, and the prefix is the string “at line.” It has a constructor for constructing when throwing an error, and an out() function for outputting to the error file. We decided to make this a struct because exception throwing was presented in class with structs. We obviously split the implementation into a .cpp file, though. We decided to pass the file name to out() instead of the constructor because this allows for more flexibility if this struct is to be used in other applications, the same Error object can output to different files.

Output handling

The clean() function in MISProgram writes an empty string to the .out and .err files. The purpose of this is so that everywhere else in the program, any output can just be appended to the .out and .err files without having to worry about the files becoming too full or not existing. Of course, we still make sure we open every file successfully before reading or writing to it, and make sure to close it when done using it.

MISProgram holds the file name of the .mis program as well as the file name of the .err and .out files because it’s faster to store these on construction instead of having to create them from the .mis program file name every time we want to output.

Loading of the .mis program

Constraints

We decided to make the .mis programs a maximum of 2048 lines. However, this is changeable by just editing one #define. Since the assignment specifies each line must be at most 1024 characters, we made the maximum file size 3MB because 2048*1024 is 2097152, which is below 3MB. This again, is changeable by editing one #define.

Loading line by line

We decided to load the program line by line because, by the assignment specifications, every variable declaration and instruction must be on separate lines, so it only makes sense to load the file in line by line.

Parsing of the parameters

Splitting the line into parameter strings

How: In the Util namespace we have a splitKeep function that splits a string into a vector of strings that contain all the parameters of the string. It splits by whitespace and commas, but stores commas in a separate element. In addition, it handles single and double quotes such that string constants are 1 element and if a string constant contains a backslash escaped quote within it, that quote is added to the 1 element for the string constant.

Why: We split the line into parameter strings to make the following steps easier. We allow whitespace between parameters because I asked on if we needed to on piazza and didn't get a response, and allowing whitespace is more flexible than not, so it's probably good to have.

Replacing backslash escape sequences.

How: find each instance of a backslash escaped sequence in the parameter strings and replace it with the corresponding character.

Why here: We replace the backslash escape sequences after splitting the line into parameter strings because we need the escape sequence while splitting to make sure any string constants with a quote within it are properly added as 1 element to the parameter strings vector.

Verifying parameters are separated by commas

How: We loop through the parameter strings and make sure every other element is a comma.

Why: We need to make sure parameters are separated by commas because that's the assignment specification.

Initializing variables

How: We initialize variables by calling the clone() method of an empty object that's in a map of type <string,Parameter*> that has a key matching the type parameter string of this variable declaration. Because we made Variable a template class, we use specialized templates to specify the initialization of each different type of variable (REAL, NUMERIC, CHAR, and STRING).

Why: This method was presented in class and its benefits are that access of maps is O(log(n)). Also, this is just an easy way of instantiating an object whose type can be identified with a key value such as a string.

Initializing instructions

How: We initialize instructions by making a new Instruction object, passing in a pointer to the corresponding function this instruction should execute. This pointer is looked up in a similar way to variables, by name in a map of type <string,InstructionFunction> where InstructionFunction is a pointer to a function following the InstructionFunction typedef format. In the initialization of instructions, we also make sure the parameters are valid and we add any necessary MicroInstructions (MicroInstructions are explained in the getting and setting of constants section below).

Why: For the same reason as it was for variables: this method was presented in class and its benefits are that access of maps is O(log(n)). Also, this is just an easy way of instantiating an object whose type can be identified with a key value such as a string. We make sure the parameters are valid here because we want to avoid having to check them at instruction execution time to enhance performance.

Storing of variables

How: We use polymorphism to store variables into a vector of type <Parameter*> in MISProgram. We can do this because Variable inherits from Parameter.

Why: We store the variables in one vector because we want to have them readily available for instructions to reference.

Storing of constants

How: We allocate and store constant when we initialize instructions with constant parameters. We allocate them in a similar way to variables, so we have a function in the Util namespace, getNumericConstant(), for returning a Parameter* to a constant specified by a parameter string. It's only for numeric constants because strings and characters only need a couple lines of code to initialize because we're just extracting them from an existing string.

Why: We store constants in this way and don't use a separate Constant class because the only consequence is one extra empty string object for every constant variable while the benefit is we don't have to handle Parameter*s being down-casted to both Variables and Constants, which would take significantly more lines of code. Why we must down-cast is explained below

Getting and setting of variables and constants

How: We get and set variables and constants by having templated, non-virtual methods, getValue() and setValue() in Parameter that down-cast itself to a Variable of the specified type and returns or sets that value.

Why: We do this because we want to use templates instead of inheritance because using templates, as explained in class, is faster than using inheritance because the function calls are resolved at compile time. We do have to know the type of the variable value before-hand, but even if we used inheritance we would still have to know the type of the variable before-hand because the variable's value type is specific to the variable. And because we only know the variable's value type at runtime because we're loading in the specifications of the variable's value type, there is no way to know at compile time which type the value is going to be. So, we store a char representing the type in every Parameter object to know which variable type to down-cast to, to get the value. While we do have to check the type at run time, we don't have to check the type at instruction execution time. Instead we check the types at load time. This allows for better performance at instruction time, but worse performance at load time. We believe this compromise is worth it considering loading takes less than half a second even for large .mis files. To avoid checking the type at instruction execution time, we use MicroInstructions to store the type specific functionality of an instruction. These MicroInstructions are stored in the Instruction object and is called within the InstructionFunction (if applicable to the instruction).

Storing of instructions

How: We store instructions in a vector of type <Instruction*> in MISProgram.

Why: We do this to have them readily available to execute and jump between.

Execution of instructions

How: We loop through each instruction and run the execute() function, which is a wrapper to its InstructionFunction pointer. As explained below jump instructions have an instruction index parameter that is the index of the instruction to jump to in the instructions vector in MISProgram. To jump, the instruction sets the JMPFlag to true and then MISProgram will set the index within the loop to 1 less than the instruction index parameter of the just executed jump instruction. It must be 1 less than because the for loop will increment it before the next iteration of the loop.

Why: This seems to be the simplest and most efficient approach to executing the instructions.

JMPs and LABELs

How: While loading in lines, if we encounter a label, we temporarily store it by name in a map of type <string, unsigned int> where the unsigned int is the index of the instruction after this label. This index is the index the instruction will be stored at in the instructions vector in MISProgram. It's an unsigned int because it will only be used as an index, and vector indexes are unsigned (avoids having to do the conversion from int to unsigned int). Also while loading in lines, if we encounter a jump instruction, we temporarily store it in a vector of type <Instruction*>. Once all lines have been loaded, the linkJumps() instruction is called, which loops through all temporarily stored jump instructions and checks if the label name parameter matches any of the temporarily stored label names. If so, it sets the instruction index parameter of the jump instruction to the instruction index corresponding to that label.

Why: As said above, this seems to be the simplest and most efficient approach to executing the instructions (specifically jump instructions).

The OUT instruction

How: The OUT instruction is passed the outFileName as a parameter, and it just outputs all of the other parameters to that outFile. The outFileName is added to the instruction at the same time instruction indexes and label names are added to jump instructions (in the initialization of the instruction).

Why: For the same reason as having the instruction index for jump instructions, this seems to be the simplest and most efficient approach to the OUT instruction. We could have made OUT separated into MicroInstructions for better performance, however that would require passing an ofstream reference to the MicroInstruction if we want to keep the file open while writing each parameter. So we decide to check the parameter types for OUT at instruction execution time to avoid having to open and close the same file up to 12 times in one OUT instruction.

MicroInstructions for Arithmetic Operations

How: We use MicroInstructions to handle arithmetic operations that take place within any instruction. The MicroInstructions are functions that do simple operations like add, sub, mul, or div one number to/by another and store the result in an accumulator. Once all arithmetic operations have been done for an instruction, the accumulators are joined into one variable and stored into the result parameter.

Why: We use MicroInstructions to avoid having to type check at instruction execution time. The benefits and consequences of this are highlighted in the getting and setting of variables and constants section above.

The rest of the instructions are fairly straightforward. The parameters you see in the .mis program are the parameters in the instruction's parameter list, and each instruction just follows the specification for what they should and shouldn't do.